手动配置数据源

package com.lilanz.microservice.cooperativeoffice.util;

import java.util.HashMap;

import javax.sql.DataSource;

import org.springframework.core.env.Environment;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.Primary;

import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

import org.springframework.orm.jpa.JpaTransactionManager;

import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;

import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;

import org.springframework.transaction.PlatformTransactionManager;

@Configuration

@EnableJpaRepositories(basePackages = "com.lilanz.microservice.cooperativeoffice.repository.SqlServer", entityManagerFactoryRef = "jpaEntityManager", transactionManagerRef = "jpaTransactionManager")

public class JpaDataSourceConfig {

  @Autowired

  private Environment env;

  @Bean

  @Primary

  public LocalContainerEntityManagerFactoryBean jpaEntityManager() {

    LocalContainerEntityManagerFactoryBean em = new LocalContainerEntityManagerFactoryBean();

    em.setDataSource(userDataSource());

    em.setPackagesToScan(new String[] { "com.lilanz.microservice.cooperativeoffice.entity.SqlServer","com.lilanz.microservice.cooperativeoffice.domain" });

    HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();

    em.setJpaVendorAdapter(vendorAdapter);

    HashMap<String, Object> properties = new HashMap<>();

    properties.put("hibernate.hbm2ddl.auto", "update");

    properties.put("hibernate.show-sql", "true");

    em.setJpaPropertyMap(properties);

    return em;

  }

  @Primary

  @Bean

  public DataSource userDataSource() {

    DriverManagerDataSource dataSource = new DriverManagerDataSource();

    dataSource.setDriverClassName(env.getProperty("tlsoft.datasource.driverClassName"));

    dataSource.setUrl(env.getProperty("tlsoft.datasource.url"));

    dataSource.setUsername(env.getProperty("tlsoft.datasource.username"));

    dataSource.setPassword(env.getProperty("tlsoft.datasource.password"));

    return dataSource;

  }

  @Primary

  @Bean

  public PlatformTransactionManager jpaTransactionManager() {

    JpaTransactionManager transactionManager = new JpaTransactionManager();

    transactionManager.setEntityManagerFactory(jpaEntityManager().getObject());

    return transactionManager;

  }

}

/\*Spring JPA 使用@CreatedDate、@CreatedBy、@LastModifiedDate、@LastModifiedBy 自动生成时间和修改者\*/

//@EntityListeners(AuditingEntityListener.class)

/\* @DynamicInsert属性:设置为true,设置为true,表示insert对象的时候,生成动态的insert语句,如果这个字段的值是null就不会加入到insert语句当中.默认false。

@DynamicUpdate属性:设置为true,设置为true,表示update对象的时候,生成动态的update语句,如果这个字段的值是null就不会被加入到update语句中,默认false。

 \*/

@DynamicInsert

@DynamicUpdate

//反序列化的时候识别是否符合本对象

private static final long serialVersionUID = 1L;

主表

@JsonManagedReference

    @OneToMany(mappedBy = "RstparkingEntity", cascade = CascadeType.ALL, fetch = FetchType.LAZY)

    private List<RstparkingDetailEntity> rsDetailList= new ArrayList<RstparkingDetailEntity>();

从表

@JsonBackReference

    @ManyToOne(cascade={CascadeType.MERGE,CascadeType.REFRESH},optional=false)//可选属性optional=false,表示父表不能为空,当删除明细,不影响父表

    @JoinColumn(name = "id")

    private RstparkingEntity rstparkingEntity;

数据库自动更新字段

/\*Spring JPA 使用@CreatedDate、@CreatedBy、@LastModifiedDate、@LastModifiedBy 自动生成时间和修改者\*/

@EntityListeners(AuditingEntityListener.class)

    @CreatedDate

    @Column(name = "gmt\_create", nullable=false,columnDefinition = " datetime DEFAULT CURRENT\_TIMESTAMP ")

    private java.util.Date gmtCreate;

    @LastModifiedDate

    @Column(name = "gmt\_modified", nullable=false,columnDefinition = " datetime DEFAULT CURRENT\_TIMESTAMP ")

    private java.util.Date gmtModified;

在SqlServer下,主键且标识列自动增量

@Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

实体父类注解

使用条件：

当我们进行开发项目时，我们经常会用到实体映射到数据库表的操作，此时我们经常会发现在我们需要隐射的几个实体类中，有几个共同的属性，例如编号ID，创建者，创建时间，修改者，修改时间，备注等。遇到这种情况，我们可能会想到把这些属性抽象出来当成一个父类，然后再以不同的实体类来继承这个父类。

那么，我们便可以使用@MappedSuperclass注解，通过这个注解，我们可以将该实体类当成基类实体，它不会隐射到数据库表，但继承它的子类实体在隐射时会自动扫描该基类实体的隐射属性，添加到子类实体的对应数据库表中。

使用环境：

1.@MappedSuperclass注解使用在父类上面，是用来标识父类的

2.@MappedSuperclass标识的类表示其不能映射到数据库表，因为其不是一个完整的实体类，但是它所拥有的属性能够隐射在其子类对用的数据库表中

3.@MappedSuperclass标识得嘞不能再有@Entity或@Table注解

@EqualsAndHashCode(callSuper = true)

@Data//默认生成的hascode是不包含父类

@EqualsAndHashCode(callSuper = true)，那就是用自己的属性和从父类继承的属性 来生成hashcode。

看Lombok的提示：

自动生成hashcode和equals方法，但是没call super，虽然这个class不是直接extends Object类，如果这是故意的，那么请加上@EqualsAndHashCode(callSuper = false) 这句话到你的model上。

@NoArgsConstructor在类上使用，它可以提供一个无参构造器，

@AllArgsConstructor同样是在类上使用，该注解提供一个全参数的构造方法，默认不提供无参构造。

**循环引用,导致json序列化失败,在各自类上加上**

@JsonIdentityInfo(generator = ObjectIdGenerators.PropertyGenerator.class, property = "id")

**@JsonIdentityInfo**

@JsonIdentityInfo也可以解决父子之间的依赖关系，但是比上面介绍的两个注解更加的灵活，在上面的两个注解中，我们自己明确类之间的父子关系，但是@JsonIdentityInfo是独立的，解决的是相互之间的依赖关系，没有父子之间的上下关系。  
使用方法如下

@Getter

@Setter

@NoArgsConstructor

@JsonIdentityInfo(property = "@id",generator = ObjectIdGenerators.IntSequenceGenerator.class)

class Boss{

String name;

String department;

//@JsonManagedReference

List<Employee> employees;

}

@Getter

@Setter

@NoArgsConstructor

@JsonIdentityInfo(property = "@id",generator = ObjectIdGenerators.IntSequenceGenerator.class)

class Employee{

String name;

//@JsonBackReference

Boss boss;

}

测试方法如下

@Test

public void JsonIdentityInfoTest() throws Exception{

CombineJacksonAnnotation.Employee employee1 = new CombineJacksonAnnotation.Employee();

employee1.setName("employee1");

CombineJacksonAnnotation.Employee employee2 = new CombineJacksonAnnotation.Employee();

employee2.setName("employee2");

CombineJacksonAnnotation.Boss boss = new CombineJacksonAnnotation.Boss();

boss.setName("boss");

boss.setDepartment("cto");

boss.setEmployees(Lists.newArrayList(employee1,employee2));

employee1.setBoss(boss);

employee2.setBoss(boss);

System.out.println(om.writeValueAsString(boss));

}

输出如下

{

"@id" : 1,

"name" : "boss",

"department" : "cto",

"employees" : [ {

"@id" : 2,

"name" : "employee1",

"boss" : 1

}, {

"@id" : 3,

"name" : "employee2",

"boss" : 1

} ]

}

其中@id表明此类的唯一标签名，我们可以使用类已经存在的属性名，如下

@Getter

@Setter

@NoArgsConstructor

@JsonIdentityInfo(property = "name",generator = ObjectIdGenerators.PropertyGenerator.class)

class Boss{

String name;

String department;

//@JsonManagedReference

List<Employee> employees;

}

@Getter

@Setter

@NoArgsConstructor

@JsonIdentityInfo(property = "name",generator = ObjectIdGenerators.PropertyGenerator.class)

class Employee{

String name;

//@JsonBackReference

Boss boss;

}

输出结果如下

{

"name" : "boss",

"department" : "cto",

"employees" : [ {

"name" : "employee1",

"boss" : "boss"

}, {

"name" : "employee2",

"boss" : "boss"

} ]

}

此注解的使用在解决循环解决的时候更加的灵活。

作者：牧羊人刘俏  
链接：https://www.jianshu.com/p/e85c3dfba052  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

Studnt实体

package com.wolfgy.domain;

import java.util.HashSet;

import java.util.Set;

import javax.persistence.CascadeType;

import javax.persistence.Entity;

import javax.persistence.FetchType;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.ManyToMany;

import org.hibernate.annotations.GenericGenerator;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

@Entity

@NoArgsConstructor

@Getter

@Setter

public class Student {

@Id

@GeneratedValue(generator = "idGenerator")

@GenericGenerator(name = "idGenerator", strategy = "uuid")

private String id;

private String sName;

@ManyToMany(cascade=CascadeType.ALL,fetch=FetchType.LAZY)

private Set<Course> courses = new HashSet<>();

}

Course实体

package com.wolfgy.domain;

import java.util.HashSet;

import java.util.Set;

import javax.persistence.CascadeType;

import javax.persistence.Entity;

import javax.persistence.FetchType;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.ManyToMany;

import org.hibernate.annotations.GenericGenerator;

import lombok.Getter;

import lombok.NoArgsConstructor;

import lombok.Setter;

@Entity

@NoArgsConstructor

@Getter

@Setter

public class Course {

@Id

@GeneratedValue(generator = "idGenerator")

@GenericGenerator(name = "idGenerator", strategy = "uuid")

private String id;

private String cName;

@ManyToMany(cascade=CascadeType.ALL,fetch=FetchType.LAZY,mappedBy="courses")//mappedBy关系由courses来维护,正常的情况下是操作student实体

private Set<Student> students= new HashSet<>();

}

#### @ManyToMany注解说明：

如代码所示，在两个实体中，我们都使用了@ManyToMany这一注解。  
这一注解表明，当前实体为多对多关系的其中一端。

注解可以在Collection、Set、List、Map上使用，我们可以根据业务需要选择。  
Collection类是Set和List的父类，在未确定使用Set或List时可使用；  
Set集合中对象不能重复，并且是无序的;  
List集合中的对象可以有重复，并且可以有排序；  
Map集合是带有key和value值的集合。

同时，我们声明的集合需要进行初始化。  
如Collection可以初始化为ArrayList或HashSet；  
Set可以初始化为HashSet；  
List可以初始化为ArrayList；  
Map可以初始化为HashMap。

在注解中，我们可以设置cascade（级联关系），fetch（加载策略）,mappedBy（声明关系的维护方）等属性。  
关于级联关系可以在我的这篇文章中了解： [==》戳这里](https://www.jianshu.com/p/e8caafce5445)  
我们简要介绍一下mappedBy。

mappedBy声明于关系的被维护方，声明的值为关系的维护方的关系对象属性名。  
在实例中，mappedBy被声明于Course类中，其值为Student类中的Set对象"courses"。即，Student为关系维护方，Course为被维护方。

但是在实际操作中，我发现其实被维护方于维护方的概念并不那么重要。**被维护方也可以对双方关系进行维护。**下面通过一组测试用例来进行说明。  
(关于mappedBy,我又更新了一篇补遗,建议阅读。阅读时间3分钟 [==》戳这里](https://www.jianshu.com/p/0ffb8ef64760" \t "_blank))  
测试用例

/\*\*

\* 仅将被维护方对象添加进维护方对象Set中

\* 保存维护方对象

\*/

@Test

public void 多对多插入1() {

Student s = new Student();

s.setSName("二狗");

Course c = new Course();

c.setCName("语文");

s.getCourses().add(c);

studentService.save(s);

}

/\*\*

\* 仅将维护方对象添加进被维护方对象Set中

\* 保存被维护方对象

\*/

@Test

public void 多对多插入2() {

Student s = new Student();

s.setSName("三汪");

Course c = new Course();

c.setCName("英语");

c.getStudents().add(s);

courseService.save(c);

}

/\*\*

\* 将双方对象均添加进双方Set中

\* 保存被维护方对象

\*/

@Test

public void 多对多插入3() {

Student s = new Student();

s.setSName("一晌");

Course c = new Course();

c.setCName("数学");

s.getCourses().add(c);

c.getStudents().add(s);

courseService.save(c);

}

/\*\*

\* 删除维护方对象

\*/

@Test

public void 多对多删除1(){

Student s = studentService.findByName("二狗");

studentService.delete(s);

}

/\*\*

\* 删除被维护方对象

\*/

@Test

public void 多对多删除2(){

//Course c = courseService.findByName("英语");

Course c = courseService.findByName("数学");

courseService.delete(c);

}

###### 测试说明及结果：

在上面的测试用例中，我们进行了三次不同的保存和三次不同的保存删除操作(多对多删除2中分别进行了两次删除操作），分别对应二狗:语文，三汪:英语，一晌:数学三组数据。

* 第一组数据（仅将被维护方对象添加进维护方对象Set中，对维护方对象的单独保存和删除）：由于操作对象是维护方，成功地在student、course以及中间表student\_courses中分别添加了数据并成功进行了删除。若将删除对象换成被维护方，同样能够成功删除。
* 第二组数据（仅将维护方对象添加进被维护方对象Set中，对被维护方对象的单独保存和删除）：操作对象在这里换成了被维护方。不负众望，出问题了。保存的时候，student表和course表倒是都成功地插入了数据，但是中间表中，并未产生对两者数据的关联。因此，在删除的时候也只删除了course中的数据。
* 第三组数据（ 将双方对象均添加进双方Set中，对被维护方对象进行保存和删除）：操作对象是被维护方，操作结果与第一组相同。

由此可知，实际操作中，只要中间表建立了关联，即使是注解定义的被维护方也是可以对双方关系进行维护的。